Aspect-Oriented Dependency Inversion

Martin E. Nordberg III
Blueprint Technologies, Inc.
7799 Leesburg Pike, Ste. 1000N
Falls Church, VA 22043
+1-703-624-4260
mnordberg@blueprinttech.com

1. ABSTRACT

The qualities of coupling and cohesion have long governed software engineering. Aspect-oriented software development (AOSD) provides a new weapon in the fight against improper coupling. For example, many OO design patterns succumb to aspect-oriented replacement when analyzed for their dependencies in terms of abstractness and stability. Component-based development (CBD) is another tool for dependency management at a larger scale. I propose combining the best of AOSD and CBD techniques for software module dependency management.

2. INTRODUCTION

Over the past several decades software engineers have improved the practice of software engineering by adding new paradigms for controlling coupling and cohesion of software modules. We have progressed through assembly language, procedural languages, structured design, object-based development, object-oriented programming, and component-based development. In this paper I examine the effects of aspect-oriented software development (AOSD) on coupling, cohesion, and dependency management.

Several principles for managing dependencies in object-oriented systems have been well articulated by Robert Martin [1]. Of the several principles he has captured in one place, three are key:

- Acyclic Dependencies Principle (ADP): Dependencies must not form cycles.
- Dependency Inversion Principle (DIP): Depend upon abstractions; do not depend upon concretions.
- Stable Dependencies Principle (SDP): Depend in the direction of stability.

Figure 1 summarizes these principles by placing classes on a coordinate system of abstractness versus instability. Well-designed classes or packages in this coordinate system fall on a diagonal from upper left to lower right [1]. Dependencies should be directed upward (DIP) and leftward (SDP).

Fair comparisons across OOP and AOP (object-oriented and aspect-oriented programming) technologies require a clear definition of dependency. Two definitions are available. The first is from the compiler's perspective: which other modules require recompilation after a developer changes one of them? The second is from the developer's perspective: which other modules likely need to be edited after editing one of them? In traditional languages the two perspectives are equivalent. For example, editing a C++ header file generally requires corollary editing and recompiling (not necessarily in that order). The equivalence disappears with the quantification (nonlinear mapping from source to binary) that is characteristic of AOSD [2]. From the compiler's perspective aspect weaving or subject composition languages fare poorly against OO alternatives precisely because of their automated weaving or composition. However, let us prefer the developer's perspective in this era of simpler languages compiled only to byte code by GHz workstations.

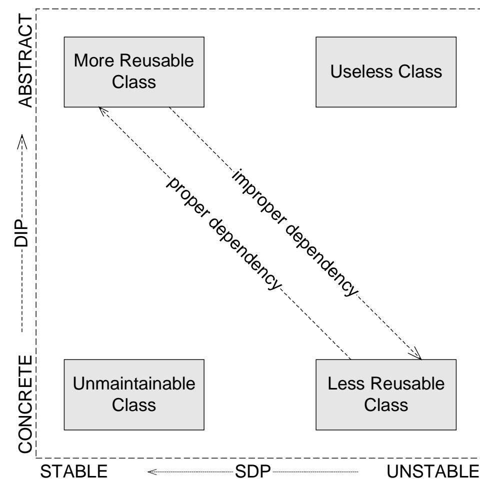

Figure 1. Plotting a design on a coordinate system of abstractness versus instability can reveal how well dependencies have been managed in that design.

In order to study the potential for aspect-oriented design to reduce or improve the dependencies in a system let's reexamine three venerable OO design patterns and aspect-oriented alternatives.

3. VISITOR

The first design pattern to succumb to aspect-oriented replacement is the Visitor pattern [3] with its infamous cyclic dependency between visitor and element hierarchy. This is illustrated in Figure 2. The improper dependency from Visitor to ConcreteElement violates all three dependency direction principles, it goes from abstract to concrete, from stable to less stable, and forms a cycle. The author has been among those who have proposed object-oriented solutions to this problem [4,5]. However, the aspect-oriented solution is basic to AOSD and much more effective (Figure 3).

Interestingly, object-oriented solutions to the cyclic Visitor dependency problem remove the "good" dependencies from Element and ConcreteElement to Visitor. The bad dependency noted in Figure 2 from Visitor to ConcreteElement is what makes the pattern what it is. In an aspect-oriented language, where an aspect can augment any class, the good dependency is in a sense left in – every class has a built in dependency on the language feature of augmentability. For example, downgrading from Aspect/J to Java might yield a poor design characterized by fat classes with low cohesion. The bad dependency, on the other

hand, disappears from programmer consciousness (there is no user source code defining what classes may be augmented), though it might be said to remain in the compiler writer's consciousness. (By the way, similar statements could have been made about polymorphism during the transition from procedural to object-oriented languages.)

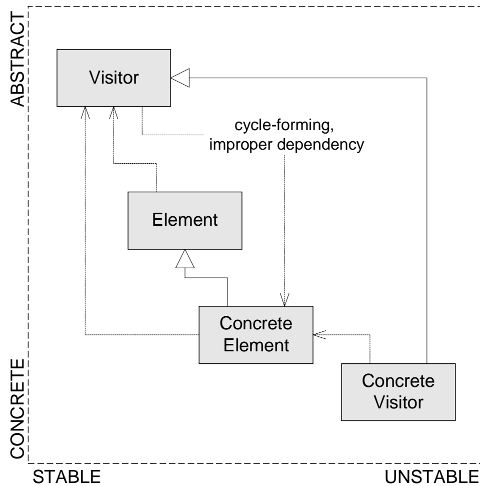

Figure 2. The Visitor pattern shows an obvious flaw when plotted on an abstractness/instability field with all dependencies explicitly shown.

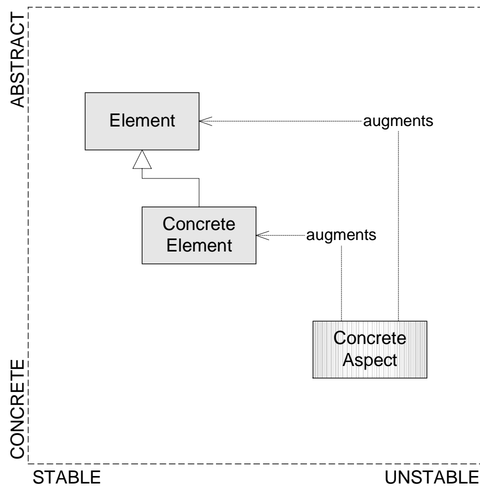

Figure 3. The aspect-oriented solution to extending the behavior of existing classes moves the "weaving" across an element hierarchy that was accomplished by the Visitor pattern from user code to programming language and compiler. This eliminates Visitor's deadly cyclic dependency from the source code.

4. OBSERVER

The case against Visitor is easy to make. More surprising is the possibility of replacing the workhorse Observer pattern [3], surprising because the Observer pattern's whole purpose is to invert dependencies. Figures 4 and 5 show the before and after of applying the Observer pattern to a temperature sensor that notifies a display when the temperature changes. The naïve solution of Figure 4 is a design in which the sensor is directly wired to the temperature display in order to trigger display changes. After adding the Subject and Observer classes of the Observer pattern (Figure 5) this hardwiring is gone, dependencies are all in the right direction, and the code for TemperatureSensor is pulled in the direction of greater reusability.

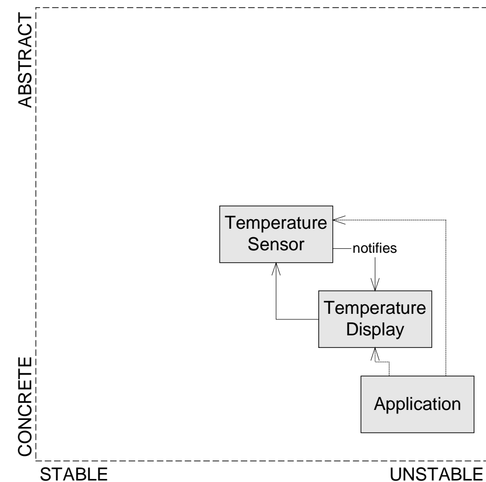

Figure 4. Without the Observer pattern a temperature sensor keeps a hard-wired reference to a temperature display in order to notify it of temperature changes. This "bad" dependency is in opposition to the ADP, DIP, and SDP.

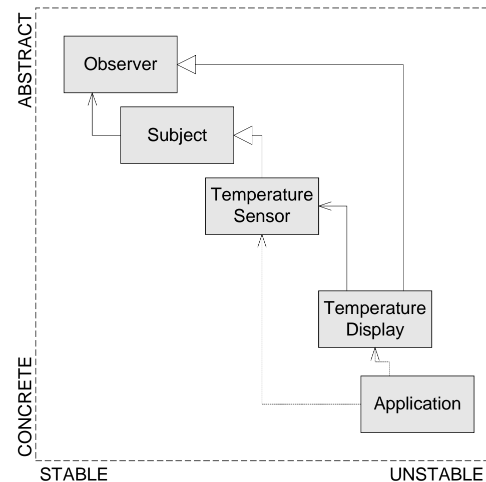

Figure 5. Adding the Observer pattern not only removes the improper dependency but also pulls some of the code in the direction of greater reusability (the upper left hand corner).

How could an aspect-oriented solution do better than this? By reducing the overall class and dependency counts and by making both TemperatureSensor and TemperatureDisplay even more reusable. Figure 6 shows the notification orchestrated by a single aspect that weaves itself into TemperatureSensor in order to notify TemperatureDisplay of changes. TemperatureSensor is more reusable because all notification code is externalised, freeing clients in different applications to employ different notification interfaces or none at all. TemperatureDisplay is more reusable because it no longer needs any knowledge of TemperatureSensor. It could easily be wired to different temperature sensors by different aspects even if one sensor has a method named getTemp() and another has a method named getTemperature().

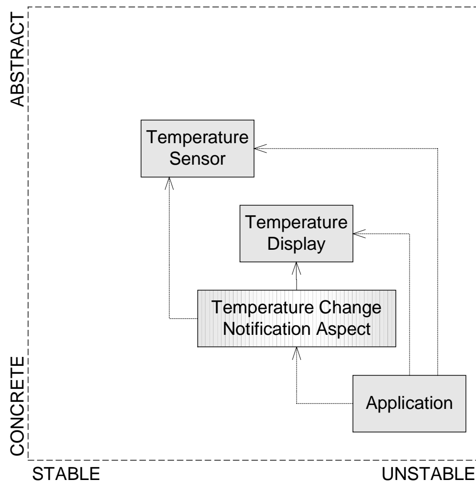

Figure 6. An aspect-oriented alternative to the Observer pattern is a notification aspect that more directly accomplishes the wiring between two objects while decoupling them completely and making each more reusable.

There is another way in which the design of Figure 6 is an improvement over Figure 5. It is a bit early to begin speaking of aspect-oriented analysis, but notice that the classes and aspects of Figure 6 more closely match the physical model that one might conceive for this application: a temperature sensor, a display, and a wire from one to the other. Object-oriented designs tend to become littered with mechanism classes, classes that serve a critical software function but have no correspondence to real world objects. With aspect-oriented notification, at least, the software module that captures notification itself has a counterpart in the real world (wire, post office, semaphores, kick in the pants, etc.). This seems like a satisfying outcome. As with the aspect-oriented answer to Visitor, the artificial mechanism classes are gone (Visitor, Observer, Subject).

In some cases the requirements for the notification might include functionality like asynchronous or idle-time firing of notifications, queuing and elimination of redundant notifications, etc. In this case the mechanism classes may be worth keeping, but still orchestrated by a notification aspect. The Aspect/J Tutorial includes an example of applying the Observer pattern in its simplest form to classes that do not in themselves provide for notifications [6]. That approach could be readily extended to more sophisticated Subject and Observer base classes.

Another advantage of aspect-oriented notification, with or without mechanism classes, is the ability to more easily generate notifications of several different types. The typical Listener mechanisms of Java Beans provide for strongly typed notifications. An aspect with optional mechanism classes can make implementing lots of different listeners straightforward.

5. EXTENSION OBJECT

The Extension Object pattern [7] combines the dependency inversion characteristics of the Observer pattern with the extensibility characteristics of the Visitor pattern. Figure 7 shows the structure of this pattern in Abstractness/Stability space. A concrete base class, Subject, provides mechanisms to dynamically add and query for extensions. A given extension must be downcast to its specific interface (SpecificExtension) for use. The pattern as a whole amounts to defining a standard way to *cross cast* from one interface reference to another.

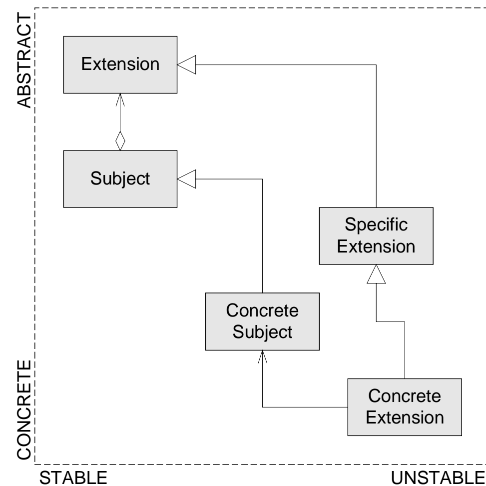

Figure 7. The Extension Object pattern includes mechanism classes (Subject and Extension) to invert the dependency between an original class and its extensions or between partial interfaces to a concrete whole.

The aspect-oriented alternative to extension objects is extension aspects (Figure 8). This approach eliminates the need for client cross casting from one extension (or base interface) to another by directly injecting the implementation of a specific interface into a class seen by a given set of clients. Besides removing the cross casting from client code, this approach reduces the run-time overhead of the cross cast and the design complexity as measured in number of classes and methods. The core functionality is more reusable since its extension mechanism is not predefined via inheritance from Subject and dependence upon Extension.

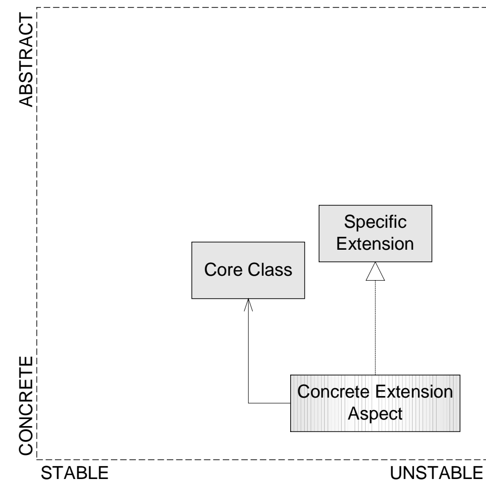

Figure 8. An aspect-oriented extension to an existing class simply augments that class to directly or indirectly realize the desired specific interface, eliminating the run-time cross-casting mechanism of the Extension Object pattern.

An interesting detail is whether the client(s) of the specific extension should be dependent upon the aspect or whether the aspect should be dependent upon the client(s). The second option would lead to a design more like the notification aspect of Figure

6 but with a wider connection between client and service. The first option is probably more appropriate if there are several clients or the clients are unknown at compile time. A judgement of the relative stability and abstractness of the client(s) and the extension they make use of would answer the question. Different approaches could even be taken for different extensions of the same core class.

6. ASPECT-ORIENTED COMPONENT CONNECTIVITY

Much of this discussion has concerned the common theme of connectivity and extensibility between a service and its clients. AOSD naturally moves extensions of base functionality to aspects and makes the core classes more stable, less concrete, and more reusable. In terms of the Open/Closed Principle (which states that modules should be open for extension and closed for modification) [1,8], classes are the closed part and aspects are the open part. The relegation of aspects to mere connectivity is an oversimplification. However, I shall make the case that connectivity is both complex and perfectly suited for aspects.

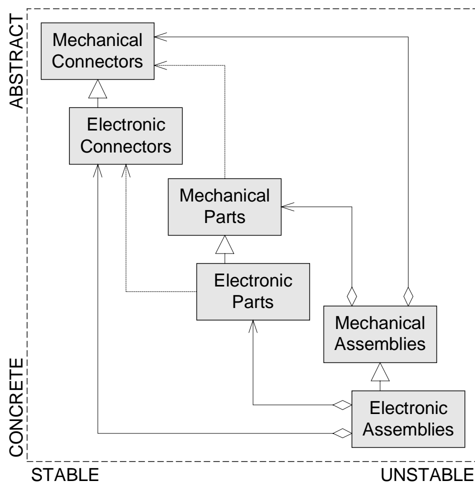

Figure 9. Mechanical and electronic assemblies have dependencies upon parts and connectors that have the proper directionality in stability and abstractness.

Several years ago software components were hailed as the integrated circuits of the software engineering profession [9]. Component technology has justifiably displaced object technology as the sharpest tool in the programmer's toolbox, but components have not achieved their promise as parts on a shelf waiting for assembly. There are many reasons, but one is that connecting software components is far more complex than connecting electronic or mechanical components. Figure 9 suggests that mechanical and electronic parts in relation to their connectors (nuts, bolts, adhesive; wires, pin patterns, bus standards) and assemblies (cars, houses; computers, VCRs) have the correct directionality of dependency. However, software connectors are more concrete and less stable than their mechanical counterparts (Figure 10). The natural dependency from software part to software connector is therefore improper according to the dependency principles. This conundrum is at the heart of many design patterns, Observer chief among them.

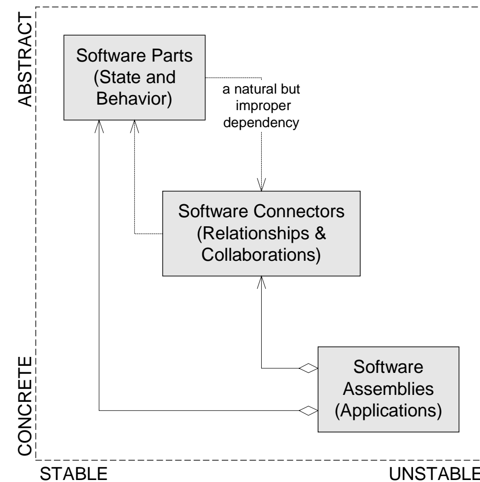

Figure 10. Unlike mechanical or electronic connectors, software connectors (static or dynamic relationships and dynamic collaborations) are more concrete and less stable. Therefore, the natural dependency of parts upon connectors is improper according to the DIP and SDP.

Why are software connectors less stable and more concrete than software parts, in contrast to their electronic or mechanical analogs? Mechanical connectors are defined by their size and shape, strength, adhesive properties, or material all of which can be standardized without much loss of design flexibility. Electronic connectors add to this list a defined behavior for the signals carried through them. This makes them less stable and abstract than their mechanical cousins. However, the variety of electrical signals (5V 66MHz digital, 12V analog, 120V 60Hz AC, etc.) is still manageably finite and well standardized. Software "signals" through software connectors, however, are as varied as software itself. I would argue that the variety of information carried between software modules is precisely what makes software so valuable.

Aspect-oriented software development has the potential to build upon all that we as a profession have learned about object-oriented and component-based development and to solve several remaining issues, like this need for an inverted dependency between part and connector. Aspect-oriented connectors can insert themselves in precise, unanticipated ways into the parts they connect. Like an imaginary engine block with no boltholes, the parts need never have been designed for a particular connection mechanism. Figure 11 shows the dependency inversion accomplished by an aspect-oriented solution that pushes connections into aspects and leaves only core functionality in classes/components/parts.

Of course, the distinction between core functionality and connectivity or extension is not always clear. In fact, Figures 10 and 11 oversimplify the abstractness and stability of software parts and connectors. More realistically, those two boxes would overlap considerably. A particular connection might be more reusable than a particular part, even though parts in general are more reusable than connections in general. The hyper slice approach has attraction on this front since functionality is all in classes sliced into different dimensions while a separate glue language defines the correspondence and composition of the

slices [10]. Other approaches make module composition a first-class citizen in aspect-oriented development [11,12].

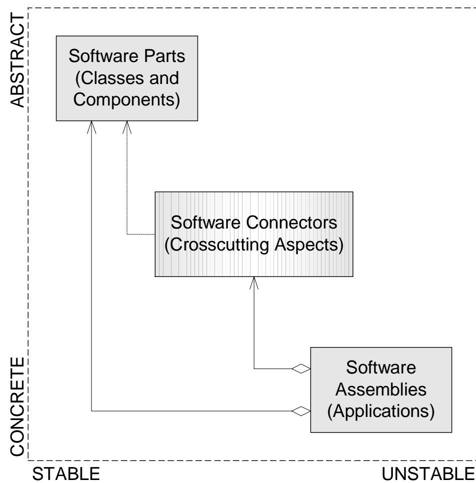

Figure 11. Placing software connectors mainly in aspects successfully inverts the dependency between software parts and software connectors.

To gain the advantages implied by Figure 11 and the foregoing discussion we must also be careful not to forget the lessons of object-oriented and component-based development (CBD) [14]. For example, in order to foster low coupling, components define narrow interfaces and completely hide their implementation. Ideally aspects would not need to peek behind the component curtain. For example, a simple-minded implementation of the notification aspect of Figure 6 could be summarized as "whenever the TemperatureSensor field myCurrentTemperature changes, call setTemperature(..) on the associated TemperatureDisplay." Besides peeking too far behind the curtain, this solution would most likely be (or worse, become) incorrect since notifications might be fired in the middle of larger operations on TemperatureSensor. A better solution might be "after completing any call to operation TemperatureSensor.readTemperature( ), call setTemperature(..) …." This is possible with current aspect-oriented languages. However, imagine a solution with even lower coupling (less knowledge of implementation details): "after completing any call to a TemperatureSensor that changes the value returned by getCurrentTemperature( ), call setTemperature(..) on the associated TemperatureDisplay." This last solution would be the most maintainable since it is coupled only to the one method, getCurrentTemperature( ), that is truly needed for the notification.

Once the directionality of dependencies between software modules is proper, it is time to work on reducing the rigidity of those dependencies. Riel [13] identifies several levels of coupling between classes. Primary among them are nil, export (interface), and overt (implementation) coupling. With AOSD it is possible to refine the list. In relationship to a class or other aspect an aspect may be:

- Independent
- Interface dependent
- Interface modifying
- Implementation dependent
- Implementation modifying

An interface modifying aspect means one that makes a new or changed interface available to clients but does not make use of existing implementation details. An implementation dependent aspect makes use of original implementation details in order to augment them (for example causing a side effect whenever a private method is called). An implementation modifying aspect changes the behavior of the original class, for example restricting changes to a member variable to leave it between 0 and 100.

The whole goal of an aspect is to augment the functionality of a core class or other aspects. However, an aspect that does so without knowledge of the unaugmented implementation is less coupled than one that freely makes use of existing implementation details. Lower coupling means better modularity even in AOSD. On the other hand, separating variable range checking from more basic computation may lead to better module cohesion with a justifiable increase in coupling. Aspects are not a license to abandon established principles of module coupling and cohesion. In some ways they require more attention to these principles rather than less.

7. AOSD + CBD = ?

The lessons to be learned in mixing aspects and components are yet to be enumerated but likely many in number.

CBD and AOSD are in some ways at odds with each other. The key practice in CBD is to take several small modules and gather them into a larger module (the component) with a narrow interface to the rest of the world. AOSD takes one or more large modules and splits them into smaller, independent, overlapping, and interlocking pieces. How can we move forward in AOSD without abandoning CBD?

I propose that one way to reconcile these competing aims is to distinguish between *functional aspects* and *connective aspects*. Functional aspects have modular multidimensional functionality as their goal and are allowed to operate only behind the component curtain. Connective aspects have configurable connectivity as their goal and operate only in the middle ground between components, without knowledge of component implementation details (Figure 12).

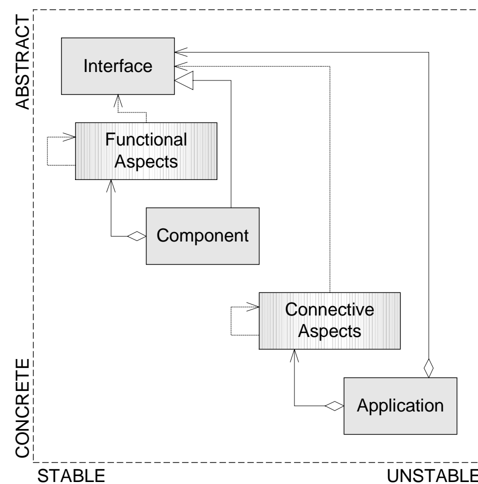

Figure 12. A distinction between functional and connective aspects might be key to retaining the benefits of component-based software development while moving forward in aspect-oriented software development.

It may now be seen that the aspects identified for replacing Visitor, Observer, and Extension Object are primarily connective aspects. Functional aspects are the more established (and more crosscutting) sort for logging, synchronization, tracing, error propagation, and so on. Functional aspects may have implementation dependencies or even implementation modifying dependencies, but connective aspects are allowed at most interface modifying dependencies. Note also that maintaining component boundaries makes physical (compiler) dependencies manageable. It may be necessary to recompile an entire component after changing one functional aspect but never an entire application.

One refinement of Figure 12 would be to make connective aspects layered for synchronization, distribution, overall coordination, and so on as in [15]. It is also possible to imagine that the design could be made recursive. For example, *components* (composed of functional aspects) might be composed by a mixture of functional and connective aspects into *services*, which are further composed by connective aspects into *applications*. Besides being buzzword compliant, and potentially unifying the several approaches to AOSD in labs worldwide, this vision paints a rosy future for AOSD in improving the practice of software engineering by building upon the best practices of today.

8. OPEN QUESTIONS

Topics for discussion during the workshop include these:

- Is AOSD in danger of forgetting the coupling/cohesion lessons of the past decades?
- Do the advantages of separation of concerns (improved cohesion) outweigh the potential disadvantages of greater coupling of smaller modules in AOSD designs? Under what circumstances?
- What novel metrics or heuristics do we need for evaluating coupling and cohesion in aspect-oriented systems?
- Does AOSD have the potential for making many OO design patterns obsolete? One objection to many design patterns is their heavy use of indirection and "lost sense of self" over multiple objects. Is there a way to avoid a different "lost sense of self" over multiple aspects? When should aspects and design patterns be combined?
- Should the software industry be attempting to standardize more reusable software connectors so that the natural dependency from part to connector is more proper? Are XML and web services steps in this direction? Are dynamic connections, e.g. service discovery, part of the solution?
- Should functional and connective aspects be orchestrated by the same language features or by different ones? What is the right mixture of imperative and declarative styles? How might fairly high-level compositional techniques be reconciled with lower level aspect-oriented languages?
- Should we keep all of OO and CBD within AOSD or are there characteristics that should be abandoned? Should the CBD curtain be ripped in some appropriate way?

9. REFERENCES

1. Martin, R.C. Design Principles and Design Patterns, http://www.objectmentor.com, 2000.
2. Filman, R.E. and Friedman, D.P. Aspect-Oriented Programming is Quantification and Obliviousness, ECOOP 2000 Workshop on Aspects and Dimensions of Concern.
3. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software*, 1995.
4. Nordberg, M.E. Default and Extrinsic Visitor. In R. Martin, D. Riehle, F. Buschmann (eds.). *Pattern Languages of Program Design 3*. Addison-Wesley Longman, Inc., 1998, 105-123.
5. Martin, R.C. Acyclic Visitor. In R. Martin, D. Riehle, F. Buschmann (eds.). *Pattern Languages of Program Design 3*. Addison-Wesley Longman, Inc., 1998, 93-103.
6. Xerox PARC, Aspect/J Primer, http://www.aspectj.org, 2001.
7. Gamma, E. Extension Object. In R. Martin, D. Riehle, F. Buschmann (eds.). *Pattern Languages of Program Design 3*. Addison-Wesley Longman, Inc., 1998, 79-88.
8. Meyer, B. Object-Oriented Software Construction, 2nd Ed., Prentice-Hall, 2000.
9. Cox, B. Planning the Software Industrial Revolution, *IEEE Software*, 1990.
10. Ossher, H. and Tarr, P. Multi-Dimensional Separation of Concerns using Hyperspaces, IBM Research Report 21452, April, 1999.
11. M. Aksit and B. Tekinerdogan, Aspect-Oriented Programming Using Composition Filters, in Object-Oriented Technology, S. Demeyer and J. Bosch (Eds.), ECOOP'98 Workshop Reader, Springer Verlag, pp. 435, July 1998.
12. Wohlstadter, E. and Devanbu, P. A Lazy Approach to Separating Architectural Concerns, ICSE Workshop on Advanced Separation of Concerns in Software Engineering, May 2001.
13. Riel, A. *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.
14. Szyperski, C. *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley Longman, Inc, 1998.
15. Navasa, A., Perez, M., and Murillo, Juan, Developing Component Based Systems Using AOP Concepts, ICSE Workshop on Advanced Separation of Concerns in Software Engineering, May 2001.